Automatic Incrementalization of Prolog based Static Analyses

Michael Eichberg Matthias Kahl Diptikalyan Saha* Mira Mezini
Klaus Ostermann

Software Technology Group, Darmstadt University of Technology
*Computer Science Department, Stony Brook University
{eichberg,kahl,mezini,ostermann}@st.informatik.tu-darmstadt.de
*dsaha@cs.sunysb.edu

Abstract. Modern development environments integrate various static analyses into the build process. Analyses that analyze the whole project whenever the project changes are impractical in this context. We present an approach to *automatic incrementalization* of analyses that are specified as tabled logic programs and evaluated using *incremental tabled evaluation*, a technique for efficiently updating memo tables in response to changes in facts and rules. The approach has been implemented and integrated into the Eclipse IDE. Our measurements show that this technique is effective for automatically incrementalizing a broad range of static analyses.

1 Introduction

Static analysis is becoming increasingly important for software developers [2]. For example, many APIs and frameworks define restrictions that cannot be expressed by function or method signatures alone. If such restrictions are not statically checked, subtle bugs can arise at runtime.[1] Enforcement of style or design guidelines, detection of bug patterns and security holes are other example areas of applying static analyses [1, 15, 19].

In this context, static analyses are most effective when they are integrated into the build process of integrated development environments (IDEs). This allows analyses to run "behind the scenes", ensuring continuous quality inspection during project development and providing the developer with immediate feedback.

However, such an integration also puts constraints on the time and space complexity of static analyses to be integrated; long build times that slow down the code-save-build cycle are unacceptable. To this end, it is desirable to compute the result of static analyses in an incremental way, whenever possible.

One option is to design an incremental version of each single static analysis. While this may be acceptable for standard analyses, it would be very inconvenient for analyses that are specific to a particular domain, framework, or company; in the latter case, it should be easy to extend the set of applicable analyses with little effort. The obligation

[1] For examples cf. Enterprise JavaBeans 3.0 Specification – Core Contracts and Requirements

to design an incremental version of each individual new analysis would be a major burden.

The work presented in this paper proposes *automatic incrementalization of static analyses* as a key technique for extensible static analysis platforms that are integrated into the incremental build process offered by modern IDEs.

We consider an analysis to be *incremental* if the following holds: Let `R` be the current result of the analysis. Then, in response to the next changes made to the code, the analysis only reprocesses those parts of the code that are necessary to compute the new result from `R`. Determining the set of software elements to reanalyze in an incremental step is not trivial: A single change might require reanalyzing multiple classes. Yet, typically this reanalyzed set represents only a small fraction of the whole project.

In our proposal, analyses are specified as Prolog programs that operate on a logic database containing a representation of the source code. New analyses can be defined declaratively, which is important for our goal of an extensible set of analyses. Specifically, we use *tabled logic programs* [5, 9, 31] which employ memoization to cache and reuse intermediate results. Tabling removes some of the shortcomings of Prolog's evaluation strategy, especially its susceptibility to infinite looping. For example, termination is guaranteed for Datalog programs (an important subset of Prolog); as such, it is suitable for a variety of static analyses.

Our basis of automatic incrementalization of static analysis is *incremental tabled evaluation* [26–30] which efficiently updates the memoized information in response to the changes in the underlying data. We use the incremental algorithm for *general logic programs* presented in [29] that is implemented on top of the tabled Prolog system XSB (ver. 2.7.1)[2]. The advantage of basing the specification and evaluation of static analyses on incremental tabled evaluation is that analyses become incremental for free, by simply declaring them as tabled. Hence, results produced by previous evaluations of analyses are automatically kept up-to-date and invalidated when needed. Incremental tabled evaluation has been tested for a few exemplary analyses (e.g. pointer analysis, push-down model checking) for C programs in [27, 28]. The work presented here generalizes and extends these preliminary results.

The contributions of this paper are as follows: FIRST, this is the first proposal to use automatic incrementalization for analyses (of Java code) that are integrated into the incremental build process of modern IDEs. To facilitate data-flow dependent analyses, a 3-address based representation in static single assignment form is used as the foundation. SECOND, we extended the capabilities of the incremental tabled evaluation algorithm. Specifically, we incorporated functionality to abolish incrementally maintained tables when they are no longer needed. THIRD, we prove the effectiveness of automatic incrementalization for a broad range of static analyses and for large changes.

The remainder of this paper is organized as follows. In Section 2, we discuss the implementation of analyses in Prolog as well as their automatic incrementalization. The section ends with an overview how the analyses are embedded into the incremental build process of Eclipse. Section 3 evaluates the proposed approach. The paper ends with the discussion of related work followed by a short summary and outlook to future work.

[2] http://xsb.sourceforge.net

```
1 package bat;
2   public class Node{  void accept(Visitor visitor){visitor.visit(this);} }
3   public class SubNode extends Node{ /* empty */ }
4
5   @Visitor(Node.class)
6   public class StructureVisitor{  public void visit(Node node){...} }
```

Listing 1.1. Sample source code

```
1  % class(PackageName,ClassName,AccessSpecifier,IsAbstract,IsFinal,SuperClass)
2  % classAnn(Class,Annotation)
3  % method(Id,DeclaringClassName,Name,AccessSpecifier,...,ReturnType,
       ListofParam,ListofAnnotations)
4
5  class('bat',ref('bat.Node'),public,false,false,ref('java.lang.Object')).
6  method(4,ref('bat.Node'),'accept',default,...,void,[parameter(ref('bat.Visitor'),[])],[]).
7
8  class('bat',ref('bat.StructureVisitor'),public,false,false,ref('java.lang.Object')).
9  classAnn(ref('bat.StructureVisitor'),annotation(type('Visitor'),value(ref('bat.Node')))).
10 method(2,ref('bat.StructureVisitor'),'visit',public,...,void,[parameter(ref('bat.Node'),[])
       ],[]).
11
12 class('bat',ref('bat.SubNode'),public,false,false,ref('bat.Node')).
```

Listing 1.2. Encoding of sample source code as Prolog database

2 Analyses in Tabled Prolog integrated into an IDE

2.1 Data Model and Prolog based Analyses

We use two example analyses to illustrate our approach to specifying static analyses as tabled Prolog queries.

The first analysis detects violations of a best practice in applying the Visitor pattern [17]. The best practice states that a visitor is expected to implement a special visit method for each type in the hierarchy it visits. The second analysis detects methods which return the self reference `this`. Such data-flow analyses are often required when implementing advanced type systems, such as, Confined Types [32].

For illustration of the analysis which detects violations of the Visitor pattern, consider the Java code in Listing 1.1. The classes `Node` (Line 2) and `StructureVisitor` (Line 6) are defined together at some point in time. Later on, the class `SubNode` (Line 3) is added to the code base. This violates the best practice, since `StructureVisitor` does not implement a `visit` method for `SubNode`. Nevertheless, the compiler will not generate any warning. A Prolog-based static analysis for detecting such a violation is shown in the following.

Listing 1.2 shows a Prolog encoding of the source code. A class fact (Line 5, 8, or 12) consists of the package name, the fully-qualified class name, the visibility, boolean

```
1  % the subtype relation is computed by invInherits and transInvInherits
2  invInherits(SuperClass,Class):- class(_,Class,_,_,_,SuperClass).
3  % transitive reflexive hull of invInherits
4  :- table transInvInherits/2.
5  transInvInherits(X,Y) :- invInherits(X,Y).
6  transInvInherits(X,X).
7  transInvInherits(X,Y) :- invInherits(X,Z), transInvInherits(Z,Y).
8
9  :- table visitor/1.
10 visitor(Class):- classAnn(Visitor,annotation(type('Visitor'),value(Node))),
11                  transInvInherits(Node,Class),
12                  not(method(_,Visitor,'visit',_,_,_,_,_,_,[parameter(Class,_)],_)).
```

Listing 1.3. Visitor Query

values denoting whether the class is final or abstract, and the name of the superclass. The first value in `method` facts (e.g. 4 in Line 6) is a generated unique identifier for a method; after that, the declaring class is specified, followed by the method's name, its visibility (`default`), an encoding of the method's modifiers using boolean values (omitted for brevity), the return type, the parameter types along with parameter annotations and the list of declared exceptions.[3]

The analysis is specified as the `visitor(Class)` query in Listing 1.3 Line 10. The query identifies visitor classes that do *not* implement a visit method for every subtype of the annotation parameter, but which are marked with the `@Visitor(Type)` annotation. For doing so, the query first selects classes with the `@Visitor` annotation to get the root of the visited hierarchy: `Node` in our example. Next, it applies the rule `transInvInherits/2` to find all classes which extend `Node`; for any such class, the query verifies that the `Visitor` has a corresponding visit method and if not the class is bound to the variable `Class`.

For each answer to the query, i.e., each binding of the variable `Class`, a warning message is generated indicating that the class violates the best practice.

The second example analysis, which checks that a method does not return the self reference (`this`), illustrates writing analyses using the 3-address based code representation in static single assignment form. A violation is shown in Line 4 on the left hand side of the following listing: `this` is assigned to the variable `o` which may be returned later on.

```
1 public Object violate(){
2   Object o;
3   if (...)
4     o = this;
5   else
6     o = null;
7   return o;
8 }
```

```
1 method(4,ref('C'),'violate',public,...).
2   if(4,2,4,...,operator,...,1).
3   label(4,3,4).
4   goto(4,4,4,2).
5   label(4,5,1).
6   label(4,7,2).
7   phi(4,8,8,p7,[phiElem(this,4),phiElem(null,1)]).
8   return(4,9,8,p7).
```

[3] All facts are properly indexed (not shown in the listing) for efficient query response.

The Prolog encoding of the method is shown on the right hand side. In general, the first value of each fact (Lines 2–8) is the id of the method and the second one is the number of the instruction. The third value is the line number of the corresponding source code — except for `labels` (Lines 3,5,6) where the third value is a method-wide unique id. The last values of `if` and `goto` statements (Lines 2,4) are the id's of labels which are the jump targets. Labels are also defined for each basic block of the control flow graph. The `phi` statement is a result of the transformation into static single assignment form and states that the value of the variable p7 (Line 7) is control flow dependent: If the id of the basic block of the last executed instruction is 4 the value of `p7` will be `this`. If the basic block's id is 1 the value will be `null`.

The query to detect the violation is shown in the Listing below. The helper predicate `initializedWithThis/2` (Lines 1,2) binds its second argument to a variable directly initialized with `this` or `this` itself. The analysis is defined in Lines 4 – 6. Line 5 binds `RetVal` to variables that are directly or indirectly initialized with `this`. Line 6 succeeds for those methods that return such a value.

```
1 initializedWithThis(MethodID, Variable) :-
2   phi(_,_,_,Variable,Phis), member(phiElem(this,_),Phis).
3
4 returnsThis(MethodID) :-
5   initializedWithThis(MethodID, Val), propagate(Val, RetVal),
6   return(MethodID,_,_,RetVal).
```

The tabled predicate `propagate/2` (Line 2,3) is the reflexive and transitive closure of all initializations of a variable; `dpropagate/2` (Line 1) implements the initialization relation.

```
1 dpropagate(V1, V2) :- phi(_,_,_,V2,Phis), member(phiElem(V1,_), Phis).
2 propagate(V,V).
3 propagate(V1,V2) :- dpropagate(V1,V3), propagate(V3,V2).
```

As shown by the `propagate/2` predicate, analyzing the data-flow is simplified as each variable is initialized exactly once and the data-flow is explicitly encoded in the `phi` facts.

2.2 Tabled Evaluation

Tabled logic programs declare certain predicates as tabled. Recursive predicates (for ensuring termination) and predicates that are reused multiple times are good candidates to be declared as tabled. Tabled resolution systems evaluate programs by memoizing subgoals of tabled predicates (referred to as *calls*) and their provable instances (referred to as *answers*) in a set of tables.

Calls are stored in a call table and all answers corresponding to a call are stored in a corresponding answer table. During resolution, if a subgoal is present in the call table, then it is resolved against the answers recorded in the corresponding answer table (*answer clause resolution*); otherwise, the subgoal is entered in the call table, its answers are computed by resolving the subgoal against program clauses (*program clause resolution*), and are entered in the answer table.

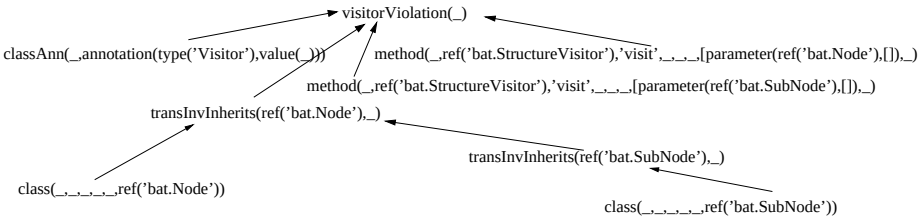

Fig. 1. Called-by Graph for Visitor Example

We exemplify the principles of tabling with the visitor example. As shown in Listing 1.3 Line 4, the recursive predicate `transInvInherits/2` is declared as tabled. Also the top level predicate `visitor/1` is declared as tabled (Line 9); a query `visitor(Class)` can be resolved by looking up the `visitor(Class)`'s answer table if the latter is non-empty. When `visitor(Class)` is executed for the first time, tabling creates an entry `visitor(Class)` in the call table and uses the rule for the `visitor` predicate to find results.

Resolving the first subgoal of the `visitor` predicate binds the variables `Node` and `Visitor` to `ref('bat.Node')` and `ref('bat.StructureVisitor')` respectively. The `transInvInherits/2` predicate is evaluated with the call `transInvInherits(ref('bat.Node'),Class)`, which is stored in the call table. The answers `Class=ref('bat.Node')` and `Class=ref('bat.Subnode')` of this call are obtained by resolution of the second clause of `transInvInherits/2`, and by resolution of the first clause of `transInvInherits/2` and `invInherits/2`, respectively. These answers are stored in the answer table of the `transInvInherits(ref('bat.Node'),Class)` call. The resolution of the last subgoal in the body of the `visitor` predicate generates only the answer `Class=ref('bat.Subnode')` for the call `visitor(Class)`, as the last subgoal fails for the substitution `Class=ref('bat.Node')`. Since `visitor/1` is tabled, any subsequent `visitor(X)` call will be resolved from its answer table.

2.3 Incremental Evaluation

Any change to a Java program causes the addition and deletion of facts to the Prolog fact base. Changes in the fact base can, in turn, render already evaluated tables stale: They may not have all the answers or the answers in the tables may be incorrect. The *non-incremental* approach to this problem is to abolish all the call and answer tables, and reissue the query. This is often wasteful, especially when the effect of the changes to the fact base is small. On the contrary, the incremental evaluation algorithm, that we use, tries to identify the calls that are *changed* and reissues only these calls. The algorithm is presented in [29] and is shortly described in the following.

A call is deemed changed iff the set of answers corresponding to the call before the change differs from that after the change. However, it is not possible to identify the set of changed calls before reevaluating any calls. Thus the incremental algorithm in [29]

over-approximates the set of changed calls by the set of *affected calls*, which are calls that can be potentially changed.

To determine the set of affected calls, the incremental algorithm maintains a data structure which keeps the dependency between calls and facts that can be changed (known as *volatile* facts). The data structure, known as *called-by graph*, is central to the incremental algorithm and is described below using our visitor example.

The called-by graph is a directed graph whose nodes consist of calls and subgoals that unify with the volatile predicates. A path from a node c_1 to node c_2 indicates that c_1 is a tabled subgoal (or a call to a volatile predicate) that was called while resolving the tabled subgoal c_2. Each edge describes the immediate dependency between calls. The graph captures the dependencies between tabled calls and calls to volatile predicates. It is first generated in the initial (non-incremental) run, and maintained over subsequent incremental runs.

The called-by graph for `visitor(Class)` is given in Figure 1. The edges from nodes `classAnn(_,annotation(type('Visitor'),value(_)))`, `transInvInherits(ref('bat.Node'),_)`, and two `method` nodes to node `visitor(_)` correspond to the first, second and two calls to the third subgoal in the body of clause `visitor(Class)`, respectively.

The incremental algorithm works in two phases: an *invalidation* phase and a *reevaluation* phase. The invalidation phase finds affected calls by bottom-up traversal of the called-by graph starting from the vertices that unify with added or deleted facts. Edges in the called-by graph are directed from callee to caller which enables us to compute the affected calls by traversing the called-by graph. For an illustration, consider the addition of a `StructureVisitor.visit(bat.SubNode)` method. This adds a fact similar to the one in Line 10 of Listing 1.2, which instead of `bat.Node` refers to `bat.SubNode`. The invalidation phase determines the `visitor(_)` call as affected, because the added fact unifies with the `method` node of the called-by graph that has `ref('bat.SubNode')` as a parameter, which, in turn, has a path to node `visitor(_)`.

If an added/deleted fact does not unify with any leaf of the called-by graph, none of the calls are affected, i.e., the change has no effect to the present set of calls and answers. For example, if we add a class `bat.Foo` that does not affect the class hierarchy of `bat.Node`, none of the existing leaves will unify with the added class fact for `bat.Foo`. Hence, none of the existing calls are affected and reevaluated. Nonetheless, a non-incremental evaluation will reevaluate all existing calls.

The specific actions taken in the invalidation phase, e.g., whether the affected calls are deleted or not, depend on the strategy of the reevaluation phase of the algorithm. However, for brevity we only describe the implemented reevaluation strategy in the following.

The algorithm approximates the changed set, called the *recomputed* set which represents the smallest set of calls that need to be reevaluated. The intuition behind the recomputed set is based on the following observations:

- Every changed call needs to be reevaluated.
- Every call that immediately depends on a changed call needs to be reevaluated (even if it itself is not changed). Note that the called-by graph contains no qualita-

tive information on *how* the change of a call affects another. Only the program has this information embedded in it and, hence, the only way to determine whether or not such a call changes, is to reevaluate it.
- If a reevaluated call is in a strongly connected component (SCC), then all calls in that SCC need to be reevaluated.

The algorithm reevaluates only the calls in the recomputed set. Two basic mechanisms are used to accomplish this:

1. Determine whether a reevaluated call is changed by comparing its answer table before and after update.
2. Evaluate the calls "bottom-up" through the called-by graph: Trigger reevaluations at higher levels only if the lower-level calls have changed.

For illustration, consider the change of the visibility modifier of `bat.SubNode`. It causes the call `transInvInherits(ref('bat.SubNode'),Class)` to be reevaluated but without any changes. Hence, the calls to `transInvInherits(ref('bat.Node'),Class)` and `visitor(Class)` are not recomputed. Thus, among the three affected tabled calls the algorithm recomputes only one call.

2.4 Deletion of Incrementally Maintained Tables

The algorithm presented in [29] incrementally maintains tables in response to changes to volatile predicates. The data structures and tables are maintained as long as the session is running. However, in our case a user can always select or deselect analyses and in case that an analysis is deselected, the maintenance of the tables that are used solely by the deselected analysis is no longer necessary. Deletion of such tables is important to reclaim unused resources and to avoid the unnecessary maintenance during incremental builds. In this paper, we therefore extend the functionality of incremental tabled evaluation to enable reclamation of incremental tables.

We provide a builtin `abolish_call(C)` which takes as the argument an incremental call C which is intended to be abolished and tries to abolish C and all calls that are directly or indirectly called by C. For example, when the visitor analysis is deselected, `abolish_call(visitor(_))` is executed. Subsequently all table space of the calls identified by `abolish_call` is reclaimed.

Below we define the set of calls that are deleted when a particular incremental call is called for deletion.

Definition 1. *Given a called-by graph $G = (V, E)$, the set $not\_deleted(C)$ defines the set of calls that should not be deleted when $abolish\_call(C)$ is called. The set $not\_deleted(C)$ is the least set satisfying the relation below:*
$C' \in not\_deleted(C)$ if

- *C is not reachable (reflexive and transitive) from C' in called-by graph*
- *$\exists C'' \in not\_deleted(C)$ and $(C', C'') \in E$ (i.e C'' depends on C').*

The set of deleted calls (denoted by $deleted(C)$) due to abolishing incremental call C is the complement of the set $not\_deleted(C)$ over all incrementally maintained calls present in the called-by graph.

We developed a called-by graph based algorithm for determining the set $deleted(C)$. The algorithm is non-trivial because of cycles in called-by graphs. It has three phases: marking, checking assumption, and deletion. The marking phase overapproximates the calls that need to be deleted and subsequent phases prune the overapproximation. Due to limited space we do not provide the algorithm here, but it can be found in an accompanying technical report [16].

2.5 IDE Integration

We have integrated the XSB Prolog engine — extended with the algorithm for incremental tabled evaluation as described in the previous sections — with the Eclipse IDE using Magellan[4]. Magellan takes care of translating every source-file of a project into its corresponding Prolog encoding. More specifically, the BAT bytecode toolkit[5] is used to convert Java class files to a 3-address based representation in static single assignment form [12] and then to convert this data into its Prolog encoding.

A full build process runs as follows: FIRST, the Prolog database is cleared, the rules used by the selected analyses are added to the database, and the Prolog facts for all Java class files are generated and added to the database. SECOND, Magellan executes the Prolog-based analyses. Each Prolog query is wrapped into a small Java class, which is responsible for (a) calling the Prolog engine to execute the query, and (b) post-processing the results, e.g., by retrieving the source code locations and by adding the error messages to Eclipse's problem view.

An incremental build maintains the database rather than rebuilding it from scratch: FIRST, whenever a document is added, changed, or removed Magellan calls the maintenance analysis and passes on the information about the edited documents. Currently, the units of change are whole classes, i.e., even when a single class' comment is modified, the maintenance analysis retracts all facts related to that class from the database and adds the class again. SECOND, the maintenance analysis then adds/removes the facts corresponding to the edited classes to/from the database and calls `update` on the Prolog database to propagate the changes to the tables. THIRD, Magellan re-evaluates the queries by simply reading the values of the corresponding tables and updates dependent IDE views such as the problems view correspondingly.

3 Evaluation

In this section, we evaluate the performance of our approach. First, the set of analyses that are used for the evaluation is presented. After that we discuss the evaluation setup and the performance figures.

3.1 Used Analyses

The analyses used for the evaluation require different kinds of information about the code and are ordered w.r.t. the extent of the required information.

[4] http://www.st.informatik.tu-darmstadt.de/Magellan
[5] http://www.st.informatik.tu-darmstadt.de/BAT

The first analysis detects classes which violate the contract defined in `java.lang.Object` stating that subclasses should always implement the `equals(...)` and `hashCode()` methods pairwise. A violation of this contract in a class `C` can lead to subtle errors when instances of `C` are stored in, e.g., `HashSet`s.

The second analysis detects covariant definitions of `equals(Object)`. Such definitions are error prone, as methods with covariant parameter definitions do not override methods defined in superclasses.

The two analyses above require only information about method signatures. The third analysis is the previously discussed analysis that checks the implementation of the Visitor design pattern [17]; hence, it requires type hierarchy information.

Finally, we added a set of 17 analyses for controlling aliasing in object-oriented systems based on confined types [32]; the basic idea is to confine the creation of aliases to a certain protection domain, in this case, to a Java package. These analyses require information about the type hierarchy and the method implementations, e.g., to analyze that a confined type is not casted to an unconfined type. Four of these analyses also require (intra-procedural) data-flow information.

3.2 Evaluation Setup and Results

The evaluation was done on a P IV, 3 Ghz with 1024 MB RAM and Sun JDK 5.

The analyzed test project is the BAT bytecode toolkit (cf. Section 2.5). This project consists of 22 packages, 790 classes, 45 interfaces, 55068 methods and approx. 395.000 facts are required to represent the method implementations. BAT contains an interface called `IStructureElement` which is implemented by 252 non-abstract classes. The visitor attached with this interface contains 504 methods. Applied on this project, the visitor query (Listing 1.3) produced 2 warnings; further, one class was identified that violated the `equals`/`hashCode` contract, and three covariant definitions of `equals` were found.

The test set was supplemented by 17 classes from a second project spread over 3 packages which implement a small part of a public key infrastructure. Initially, confined types were used in two of the packages. When performing the changes described in Table 1, classes in the third package were also made confined. Initially, 17 different errors related to confined types were in the code.

In case of a full build, the time to create the Prolog facts takes 3300 msecs. This includes the transformation of the Java files into the 3-address representation and the creation of the Prolog facts; to add the generated facts to the database, XSB requires another 5200 msecs. Since all tables are initally empty, the first evaluation of the queries takes 328 msecs.

Changes shown in Table 1 were executed in the given order. The first eight changes affect core classes of the BAT project by modifying a comment, adding a field, adding a method, renaming a class file, or adding a new class. Changes 9 to 16 simulate the use of confined types in the project by marking classes as confined or unconfined.

To better assess the effect of a change, the number of affected classes is shown in the third column of Table 1; further, the number of methods defined by the classes and the total number of facts that were removed and added is given. In the fourth column, the results produced by the queries after performing the changes is given: The first is

Run	Description of the Changes	removed / added Classes, Methods, Facts	Results	msecs. no incr. XSB	msecs. incr. XSB
1	inserted two empty lines into a class	1, 504, 2779 / 1, 504, 2779	1/3/2/17	673	390
2	deleted a small method and the implementation of another small method	1, 504, 2779 / 1, 503, 2771	1/3/3/17	627	390
3	created a new field along with the corresponding getter method; further a new empty method is created in a different class	2, 41, 430 / 2, 43, 447	1/3/3/17	468	156
4	ten fields and corresponding getters and setters are created	1, 9, 40 / 1, 29, 141	1/3/3/17	454	78
5	refactored the name of a class which has 6 children; hence 7 classes are affected	7, 112, 2690 / 7, 112, 2690	1/3/3/17	812	578
6	added a blank into the comment of a small class	1, 9, 41 / 1, 9, 41	1/3/3/17	437	78
7	deleted six small methods as a whole and also deleted the content of another six methods	1, 503, 2771 / 1, 497, 2723	1/3/9/17	669	390
8	added a new class which implements an interface	0, 0, 0 / 1, 3, 11	1/3/10/17	406	63
9	added a new method which leads to a violation of a widening constraint	1, 3, 15 / 1, 4, 19	1/3/10/18	389	48
10	added a new method	1, 3, 15 / 1, 4, 18	1/3/10/18	392	78
11	deleted the method which was added in the previous change	1, 4, 18 / 1, 3, 15	1/3/10/18	405	78
12	changed the superclass, modified a small method, added a new field and a another small method which violates a widening constraint	1, 3, 15 / 1, 4, 31	1/3/10/25	453	141
13	created a new interface which is implemented by two classes, deleted parts of a method	3, 9, 72 / 2, 9, 65	1/3/10/25	454	172
14	declared a class as confined	1, 5, 32 / 1, 5, 32	1/3/10/24	454	141
15	changed the implementation of a method	1, 4, 19 / 1, 4, 22	1/3/10/24	469	78
16	a new field is added to three different confined classes	3, 10, 69 / 3, 10, 73	1/3/10/27	500	187
in average				*ø503,8*	*ø190,4*

Table 1. Change description and timing results

the number of violations of the `equals`/`hashCode` contract, the second is the number of covariant `equals` methods, the third is the number of violations related to the `Visitor` design pattern, and the fourth is the number of violations of confinement rules.

The last two columns of Table 1 compare the time required to update the database and to retrieve the new set of results using tabling without incremental evaluation of XSB and using tabling with incremental evaluation.

The numbers for incremental evaluation (last column in Table 1) result from summing the time for removing and adding facts with the time for incrementally maintaining the tables. All queries are evaluated in roughly 0 msecs independent of the code change that triggers the evaluation. This is because query results are tabled and the extraction of the answers from a table only depends on the number of identified errors. With non-incremental evaluation, the time to execute the queries also remains constant at roughly 300 msecs. The difference between this 300 msecs and the numbers in the corresponding column in Table 1 is spent to add/remove facts.

To summarize, we the draw following conclusions: FIRST, the approach is fast enough to execute a reasonable number of analyses along with the incremental build process for projects with at least 1000 classes; executing all discussed analyses simultaneously is feasible. Even in case of changes that affect large numbers of facts (Runs 1,2,5,7) the execution times are acceptable. SECOND, in comparison to non-incremental evaluation, our system is between 1.4 and 8 times faster. In case of non-incremental evaluation, the queries need to be reevaluated from scratch after every change; in particular it is necessary to explicitly delete all tables, as the tables are not maintained incrementally. THIRD, most of the time required by the incremental build goes to maintain the tables. This time is largely dependent on the number of facts that need to be removed and added. Hence, if the granularity of a change would be more fine-grained than an entire class, the overall time could be further improved.

4 Related Work

Writing analyses using a logic language, such as, Prolog is not new. Many classical program analysis problems can be readily encoded into deductive frameworks [13] and various practical implementations have been stemmed based on such encodings. E.g., Besson and Jensen [3] discuss the implementation of a class analysis using Datalog.

Various approaches use declarative query languages to implement static analyses [34, 14, 21]. For example, the Program Query Language (PQL) presented in [23] allows programmers to express queries in application specific context and allows them to specify actions along with the queries. PQL is then transformed into Datalog which is evaluated using the BDD based evaluation framework BDDBDDB. Soul [35] is a logic meta-language implemented in Smalltalk to express and extract structural relationships (Prolog like) in class-based object-oriented systems. ASTLOG [11] is also a Prolog like language to identify bug patterns primarily in C/C++ code. ASTLOG directly operates on top of the source syntactic structures to get a better performance when compared with using a Prolog database. Spine [4] is a typed first-order logic similar to Prolog for describing design patterns and their constraints. Given a Spine specification of a design

pattern the Hedgehog proof system [4] is then used to reason about the implementation of design patterns in Java. However, none of the above techniques supports incrementalization, i.e., in case of small changes to the source code, all analyses have to be repeated for the whole program to get an up-to-date view.

CodeQuest [18] uses Datalog for querying code. Unlike the above approaches, it realizes the importance of incremental updates. CodeQuest incrementally maintains the database of facts. When notified by the Eclipse platform about a change to a compilation unit, CodeQuest removes from the database all facts that are directly or indirectly related to the compilation unit (determined using ad-hoc stored procedures); it then reparses the compilation unit and populates the database with the new facts. Compared to CodeQuest, our approach also employs incremental maintenance of query results.

The problem of incremental evaluation has been addressed in various fields of research, such as view maintenance in databases, model checking, program analysis, logic programming, functional programming, attribute grammar evaluation, and AI. In the focus of this discussion is only the problem of incremental evaluation in the area of program analysis. Most of the existing work addressing incremental evaluation in the latter area is catered toward particular kinds of static analyses, e.g., pointer analysis, data-flow analysis, MOD analysis, and verification of safety properties, and cannot be readily generalized to a wide range of analyses.

An incremental alias analysis is presented in [37] which is based on Landi-Ryders's flow- and context-sensitive alias analysis [20]. A variety of incremental algorithms have been developed for data flow analysis problems. Some of them use the elimination method [6, 8, 25]; others are based on restarting iterations [24], while both techniques are combined in [22]. A comparison of incremental iterative algorithms for data flow analysis can be found in [7]. The effectiveness of incremental analysis has been shown for MOD analysis of C programs [36]. Pollock and Soffa [24] presented a precise incremental iterative algorithm using change classification and reinitialization for bitvector problems. In [10], an algorithm is presented that incrementally analyzes the verification of safety properties of a program. In [33], an incremental algorithm is presented which analyzes part of the program assuming no previous analysis result. This algorithm monitors the analysis results incrementally in each phase to direct the analysis in those parts of the program which offer the highest expected optimized return. This work does not consider the problem of updating existing analysis results to reflect the effect of program changes.

The above approaches to incremental evaluation of static analysis are specific to the analysis considered and the used techniques are not easy to generalize for incremental evaluation of other static analyses. The first step toward developing techniques for automatic incrementalization of a broad range of analysis is the work by Saha and Ramakrishnan on incremental evaluation of tabled logic programs [26, 28, 30]. Tabled logic programs offer a declarative way of encoding a large variety of program analysis [13]. As discussed in this paper, incremental tabled evaluation offers a generic approach to incrementalizing static analysis.

5 Summary and Future Work

In this paper, we proposed to use incremental tabled Prolog for the automatic incrementalization of static analyses. This enables developers to write static analyses with the full-build case in mind. The analyses are automatically incrementalized by the Prolog engine, i.e., in case of changes to the fact-base only the necessary parts of the project are reanalyzed. The analyses are implemented on top of a 3-address based representation in SSA form. This representation proved to be well-suited for intra-procedural data-flow analyses and enables an efficient implementation of static analyses.

The automatic incrementalization frees the developer from the burden of developing incremental algorithms for each single analysis and, thus, facilitates the development of new domain and project specific static analyses. As shown in the evaluation section, the proposal significantly improves the performance of static analyses compared to their non-incremental versions and enables to tightly integrate them with the incremental build process of an IDE.

Further performance improvements will be in the focus of future work. One possibility to improve performance is by decreasing the change granularity, which is currently at the class level. By pushing the granularity down, e.g., to the level of instructions, further overall performance improvements are expected.

References

1. K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the Symposium on Security and Privacy*. IEEE, 2002.
2. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of IFM*. Springer, 2004.
3. F. Besson and T. P. Jensen. Modular class analysis with datalog. In *Proceedings of SAS*. Springer, 2003.
4. A. Blewitt, A. Bundy, and I. Stark. Automatic verification of java design patterns. In *Proceedings of ASE*. IEEE, 2001.
5. R. Bol and L. Degerstadt. Tabulated resolution for well-founded semantics. In *Proceedings of ILPS*. MIT Press, 1993.
6. M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *TOPLAS*, 12(3), 1990.
7. M. G. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7), 1990.
8. M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. In *Proceedings of POPL*. ACM, 1988.
9. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1), 1996.
10. C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *Proceedings of CAV*. Springer, 2005.
11. R. F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of DSL*. USENIX, 1997.
12. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 1991.

13. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *Proceedings of PLDI*. ACM, 1996.
14. M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. In *Proceedings of WCRE*. IEEE, 2004.
15. M. Eichberg, T. Schäfer, and M. Mezini. Using annotations to check structural properties of classes. In *Proceedings of FASE*. Springer, 2005.
16. Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. Automatic incrementalization of static analyses. Technical report, 2006. (http://www.st.informatik.tu-darmstadt.de/Magellan).
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
18. Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of ECOOP*. Springer, 2006.
19. D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12), 2004.
20. W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of PLDI*. ACM, 1992.
21. Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *Proceedings of PLDI*. ACM, 2004.
22. T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proceedings of POPL*. ACM, 1990.
23. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of OOPSLA*. ACM, 2005.
24. L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12), 1989.
25. B. G. Ryder and M. C. Paull. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1), 1988.
26. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *Proceedings of ICLP*. Springer, 2003.
27. D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of PPDP*. ACM, 2005.
28. D. Saha and C. R. Ramakrishnan. Symbolic support graph: A space-efficient data structure for incremental tabled evaluation. In *Proceedings of ICLP*. Springer, 2005.
29. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled prolog: Beyond pure logic programs. In *Proceedings of PADL*. Springer, 2006.
30. D. Saha and C. R. Ramakrishnan. A local algorithm for incremental evaluation of logic programs. In *Proceedings of ICLP*. Springer, 2006.
31. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Proceedings of ICLP*. Springer, 1986.
32. J. Vitek and B. Bokowski. Confined types in java. *Software Practice and Experience*, 31(6), 2001.
33. F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of PLDI*. ACM, 2001.
34. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of PLDI*. ACM, 2004.
35. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA*. IEEE, 1998.
36. J. Yur, B. G. Ryder, W. Landi, and P. Stocks. Incremental analysis of side effects for C software system. In *Proceedings of ICSE*. ACM, 1997.
37. J. Yur, B. G. Ryder, and W. A. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of ICSE*. IEEE, 1999.